Міністерство освіти і науки України
Харківський національний університет імені В. Н. Каразіна Факультет комп'ютерних наук

Курсова робота
з дисципліни «Теорія алгоритмів»
на тему:
«АА-Дерево»

Виконав:
студент групи КС - 21
Солотопов К. С.

Харків – 2022

ЗМІСТ

РОЗДІЛ 1 ВСТУП: ОБ'ЄКТ, ПРЕДМЕТ, МЕТА

Об'єкт - АА-дерево, додавання/видалення вузла в АА-Дерево, операція skew, операція, split.

Предмет – структура даних АА-дерево, операції АА-дерева.

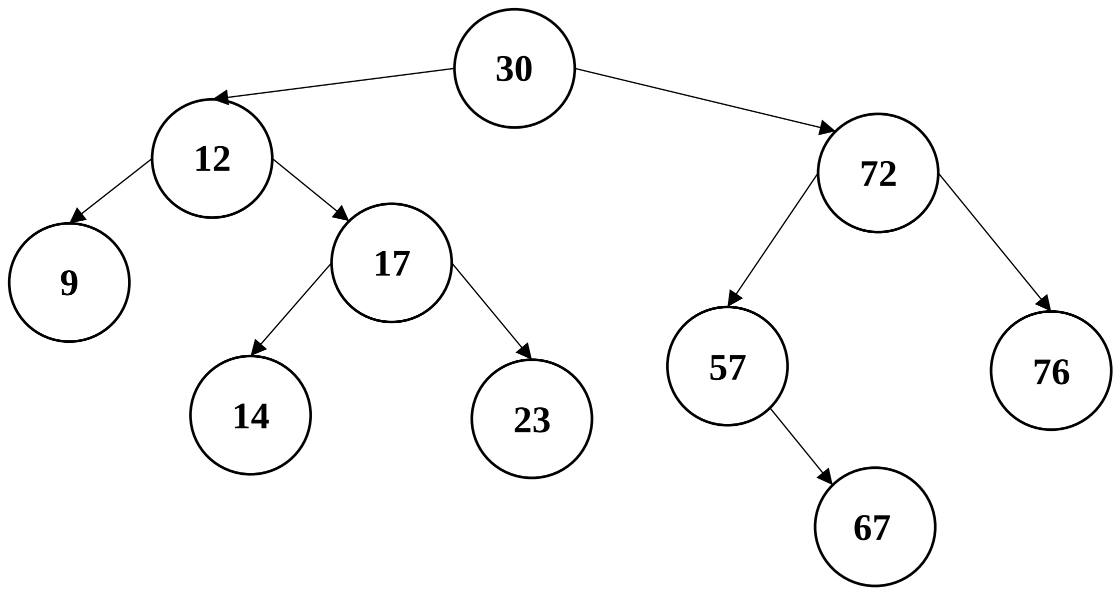
Мета – ознайомитись з АА-Деревом та використовуючи мову програмування Java реалізувати цю структуру даних. Розглянути відмінність від Червоно-Чорного Дерева та методи самобалансування структури.

РОЗДІЛ 2 ТЕОРЕТИЧНІ ВІДОМОСТІ ТА АНАЛІЗ АЛГОРИТМУ

2.1 Загальне визначення АА-дерева та його візуалізація

АА-дерево — структура даних, що є збалансованим бінарним деревом пошуку, яке є різновидом Червоно-Чорного Дерева.

АА-дерево названо на честь Арне Андерссона, який і запропонував у 1993 році цю модифікацію Червоно-Чорного Дерева, в основу якої поклав поняття рівня вершини.



Малюнок 2.1 – Побудоване АА-Дерево

2.2 Властивості АА-Дерева

Рівень вершини – вертикальна висота відповідної вершини, яка представлена у числовому полі (див. Лістинг 2.1)

Визначення 2.1

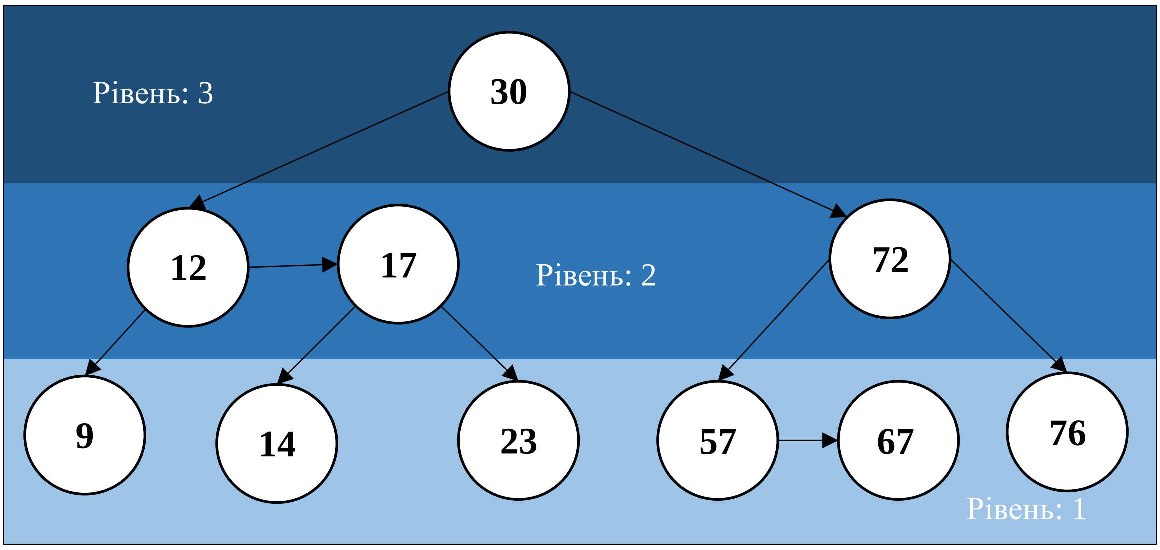
```
public class Node {
    public int data;

    int level; // Рівень вершини
    Node left; // Лівий нащадок
    Node right; // Правий нащадок
    Node parent; // Предок
    ...
}
```

Лістинг 2.1 – поля класу Node для представлення вузла в АА-Дереві

Для АА-Дерева існує основне правило, яке несе в собі реалізація цієї структури: до однієї вершини можна приєднати іншу вершину того ж рівня, але тільки одну і лише праворуч.

Але варто розглянути правила цілком:

- Рівень кожного листа дорівнює 1.
- Рівень кожної лівої дитини рівно на один менший, ніж у батька.
- Рівень кожної правої дитини дорівнює або на одну меншу, ніж у її батька.
- Рівень кожного правого онука значно менший, ніж у його прабатька.
- Кожна вершина з рівнем понад 1 має двох дітей.

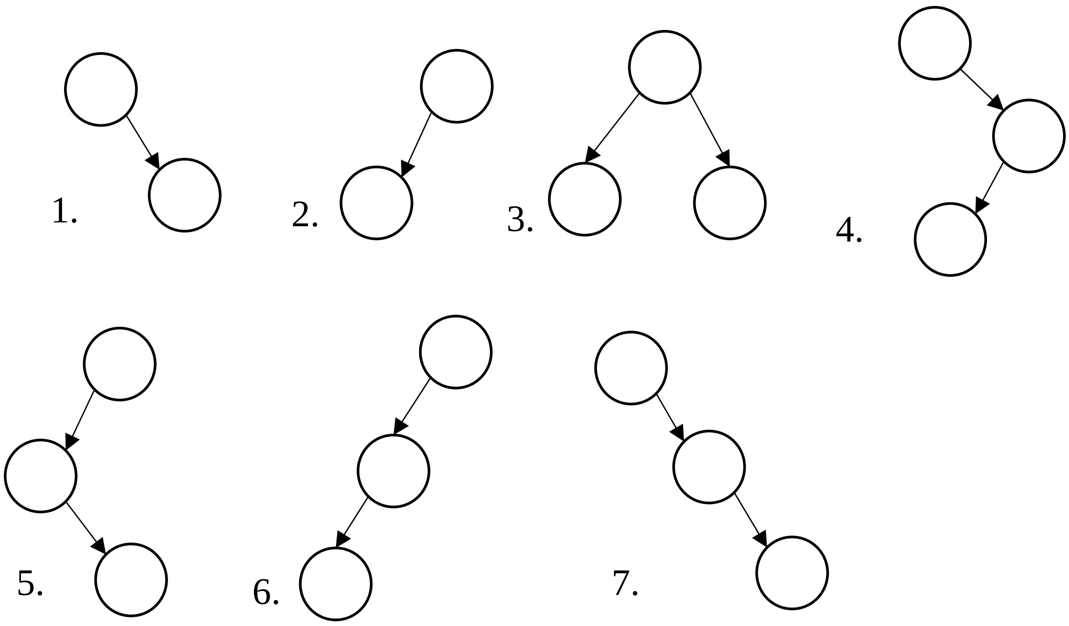


Малюнок 2.2 – Візуалізація властивостей АА-Дерева

2.3 Відмінності від Червоно-Чорного Дерева

Виходячи з Визначення 2.1, червоні вершини в АА-Дереві можуть бути додані тільки як праві нащадки. Це призводить до моделювання 2-3 дерева замість 2-4 дерев. Це спрощує операції обслуговування структури.

У той час як Червоно-Чорному Дереву потрібно обробити сім різних варіантів розташування вершин:



Малюнок 2.3 - Опрацьовані варіанти для балансування Червоно-Чорного Дерева

АА-Дереву потрібно до розгляду всього два варіанти:

Малюнок 2.4 – Перевірка на лівий горизонтальний зв'язок та на два послідовних правих горизонтальних зв'язків

Властивості Червоно-Чорного Дерева – у додатку А

2.4 Ефективність АА-Дерева

Оцінка на висоту АА-дерева рівноцінна оцінці для Червоно-Чорного Дерева: 2·log2(N) – оскільки АА-Дерево зберігає структуру Червоно-Чорного дерева. Відповідно, складність всіх операцій O(logN) – у збалансованому двійковому дереві пошуку багато операцій реалізуються за O(h).

Ефективність пошуку.

Ми йдемо шляхом від кореня до шуканого вузла (або до NIL-аркуша). На кожному рівні ми виконуємо порівняння. Таким чином, вартість пошуку пропорційна висоті дерева.

Позначимо через n кількість вузлів дерева. За однією з властивостей Червоно-Чорного Дерева відомо, що найдовший шлях не більш ніж удвічі довший за найкоротший шлях. Звідси випливає, що висота дерева обмежена O(log n).

Таким чином, тимчасова складність пошуку вузла в AA-tree становить: O(log n).

Ефективність вставки

При вставці спочатку виконуємо пошук, вартість якого визначена вище. Далі вставляємо вузол. Вартість цього завжди незалежно від розміру дерева, тому O(1). Потім перевіряємо правила АА-Дерева і за необхідності відновлюємо їх (skew, split – див. розділ 2.4). Ми робимо це, з вставленого вузла та сходячи до кореня. Кожна з цих операцій сама собою має постійний час O(1). Таким чином, загальний час перевірки та ремонту дерева також пропорційний його висоті.

Таким чином, тимчасова складність вставки в червоно-чорне дерево також дорівнює: O(log n)

Ефективність видалення

Як і при вставці, ми спочатку шукаємо вузол, що видаляється за час O(log n). Крім того, вартість видалення не залежить від розміру дерева, тому вона стала O (1).

Аналогічно вставці перевіряємо правила АА-Дерева та за необхідності відновлюємо їх (skew, split – див. розділ 2.4). Ми робимо це, з вставленого вузла та сходячи до кореня. Кожна з цих операцій сама по собі має постійний час O(1). Таким чином, загальний час перевірки та ремонту дерева також пропорційний його висоті.

Таким чином, тимчасова складність видалення з червоно-чорного дерева також дорівнює O(log n)

Додаткові витрати пам'яті

Швидкість роботи АА-дерева еквівалентна швидкості роботи червоно-чорного дерева, але оскільки в реалізації замість кольору зберігають рівень вершини, додаткові витрати пам'яті досягають байта (у Червоно-Чорному дереві використовують true/false – чорний/червоний колір посилання).

РОЗДІЛ 3 ПРОГРАМНА РЕАЛІЗАЦІЯ

3.1 Балансування АА-Дерева

Горизонтальне ребро - ребро, що з'єднує вершини з однаковим рівнем (аналогічно червоній ссилці у Червоно-Чорному Дереві).

Визначення 3.1

Як стало відомо з розділів 2.3 і 2.4 в АА-Дереві можуть бути лише праві ребра, що не йдуть поспіль, і не можуть бути всі ліві горизонтальні ребра. Ці жорсткіші обмеження, аналогічні обмеженням на Червоно-Чорних Деревах, призводять до більш простої реалізації балансування АА-Дерева.

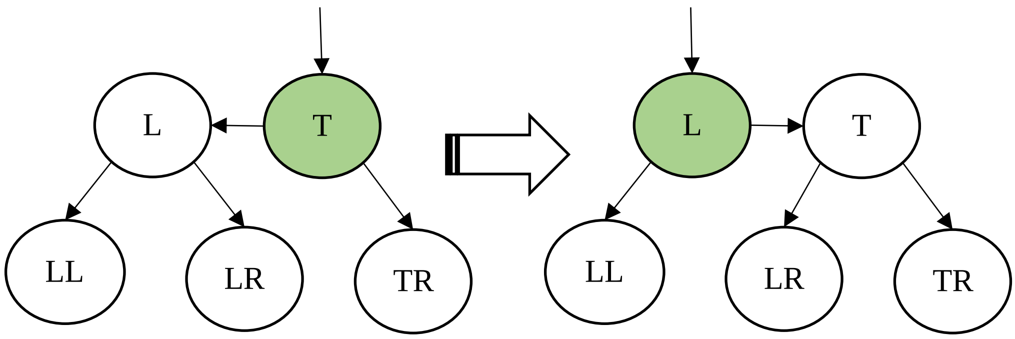
Для цього передбачені дві наступні операції: skew та split (які будуть розглянуті докладніше у розділах 3.1.1 та 3.1.2 відповідно).

3.1.1 Метод балансування Skew

Метод приймає вузол, що представляє АА-Дерево, яке необхідно перебалансувати. Потім він усуває ліве горизонтальне ребро: відтворюється праве обертання, щоб замінити поддерево, що містить лівий горизонтальний зв'язок, на поддерево, що містить правий горизонтальний зв'язок. Повертає вузол, що представляє перебалансоване дерево.

```
private Node skew(Node node) {
    if (node == nil) {
        return nil;
    } else if (node.left == nil) {
        return node;
    } else if (node.left.level == node.level) {
        Node leftChild = node.left;
        node.left = leftChild.right;
        leftChild.right = node;
        return leftChild;
    } else {
        return node;
    }
}
```

Лістинг 3.1 - Реалізація Skew



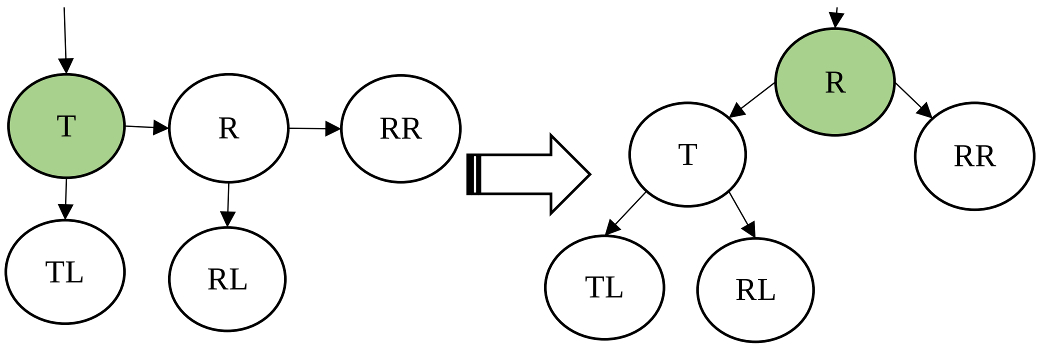
Малюнок 3.1 – Ілюстрація роботи Skew

3.1.2 Метод балансування Split

Метод приймає вузол, що представляє АА-Дерево, яке необхідно перебалансувати. Потім він усуває два послідовні правих горизонтальних ребра: відтворюється ліве обертання і збільшення рівня вузла, щоб замінити піддерево, що містить два або більше послідовних горизонтальних зв'язків, на вершину, що містить два піддерева з меншим рівнем. Повертає вузол, що репрезентує перебалансоване дерево.

```
private Node split(Node node) {
    if (node == nil) {
        return nil;
    } else if (node.right == nil || node.right.right == nil) {
        return node;
    } else if (node.level == node.right.right.level) {
        Node rightChild = node.right;
        node.right = rightChild.left;
        rightChild.left = node;

        rightChild.level = rightChild.level + 1;
        return rightChild;
    } else {
        return node;
    }
}
```

Лістинг 3.2 – Реалізація Split



Малюнок 3.2 – Ілюстрація роботи Split

3.2 Вставка в АА-Дерево

Метод набуває значення для вставки і кореневий вузол дерева, яке потрібно її відтворити. Вставка починається зі звичайної процедури пошуку та вставки, як у двійковому дереві. Потім, у міру розкручування стека дзвінків, легко перевірити правильність дерева та за необхідності виконати будь-які повороти. Якщо виникає ліве горизонтальне посилання, виконується метод skew, а якщо виникають два горизонтальні праві посилання, виконується метод split, можливо, збільшуючи рівень нового кореневого вузла поточного піддерева. Повертає вузол, який репрезентує збалансоване дерево з новим вставленим вузлом.

```
private Node insertNode(int data, Node node) {
    if (node == nil) {
        node = new Node(data, nil, nil);
    } else if (data < node.data) {
        node.left = insertNode(data, node.left);
    } else if (data > node.data) {
        node.right = insertNode(data, node.right);
    } else {
        return node;
    }

    node = skew(node);
    node = split(node);
    return node;
}
```

Лістинг 3.3 – Реалізація вставки

3.3 Видалення з АА-Дерева

Метод набуває значення видалення і кореневий вузол дерева, у якому потрібно його відтворити. Як і більшості збалансованих бінарних дерев, видалення внутрішнього вузла можна перетворити на видалення листового вузла, замінивши внутрішній вузол або його найближчим предком, або нащадком. Отримання предка - це просто перехід по одному лівому засланню, а потім по всіх правих посиланнях, що залишилися. Так само можна знайти нащадка, пройшовши один раз праворуч і ліворуч, поки не буде знайдено нульовий покажчик. Через властивості АА-Дерева всіх вузлів рівня вище 1, що мають два дочірні вузли, вузол-нащадок або вузол-предок буде перебувати на рівні 1, що робить їх видалення тривіальним.

Після видалення першим кроком до підтримки достовірності дерева є зниження рівня будь-яких вузлів, чиї дочірні елементи знаходяться на два рівні нижче за них або у яких відсутні дочірні вузли. Потім весь рівень потрібно прогнати через skew і split. Ускладнення в тому, що методу потрібно використовувати skew та split на весь рівень.

```java
private Node deleteNode(int data, Node node) {
    if (node == nil) {
        return node;
    } else if (data > node.data) {
        node.right = deleteNode(data, node.right);
    } else if (data < node.data) {
        node.left = deleteNode(data, node.left);
    } else {
        if (node.left == nil && node.right == nil) {
            return node.right;
        } else if (node.left == nil) {
            Node temp = node.right;
            while (temp.left.data != 0) {
                temp = temp.left;
            }
            node.right = deleteNode(temp.data, node.right);
            node.data = temp.data;
        } else {
            Node temp = node.left;
            while (temp.right.data != 0) {
                temp = temp.right;
            }
            node.left = deleteNode(temp.data, node.left);
            node.data = temp.data;
        }
    }
    node = decreaseLevel(node);
    node = skew(node);
    node.right = skew(node.right);

    if (node.right != nil) {
        node.right.right = skew(node.right.right);
    }
    node = split(node);
    node.right = skew(node.right);
    return node;
}
```

Лістинг 3.4 – Реалізація видалення

```java
private Node decreaseLevel(Node node) {
    int shouldBe = Math.min(node.left.level, node.right.level) + 1;
    if (shouldBe < node.level) {
        node.level = shouldBe;
        if (shouldBe < node.right.level) {
            node.right.level = shouldBe;
        }
    }
    return node;
}
```

Лістинг 3.5 - Реалізація зниження рівня

3.4 Базові операції Бінарного Дерева

Такі операції як: пошук вузла, inorder обхід дерева, preorder обхід дерева, postorder обхід дерева і підрахунок вузлів не буде розглядатися, оскільки ці операції ідентичні реалізації звичайного Бінарного Дерева.

Код цих операцій подано у Додатку В.

РОЗДІЛ 4 РЕЗУЛЬТАТИ РОБОТИ ПРОГРАМИ

```
PROGRAM START
1 - INSERT
2 - DELETE
3 - SEARCH
4 - 2D OUTPUT
5 - INORDER
6 - PREORDER
7 - POSTORDER
8 - NUMBER OF NODES
9 - CLEAR TREE
0 - EXIT
ENTER: 1

Enter data for insert: 30

1 - INSERT
2 - DELETE
3 - SEARCH
4 - 2D OUTPUT
5 - INORDER
6 - PREORDER
7 - POSTORDER
8 - NUMBER OF NODES
9 - CLEAR TREE
0 - EXIT
ENTER: 1

Enter data for insert: 72

1 - INSERT
2 - DELETE
3 - SEARCH
4 - 2D OUTPUT
5 - INORDER
6 - PREORDER
7 - POSTORDER
8 - NUMBER OF NODES
9 - CLEAR TREE
0 - EXIT
ENTER: 1

Enter data for insert: 12
```

Скріншот 4.1 – Приклад вставки

На скріншоті 4.1 у дерево було вставлено 3 значення. Опустимо демонстрацію подальшої вставки. Маємо в АА-Дереві вузли з такими значеннями: 17 12 9 14 30 23 72 57 67 76.

```
1 - INSERT
2 - DELETE
3 - SEARCH
4 - 2D OUTPUT
5 - INORDER
6 - PREORDER
7 - POSTORDER
8 - NUMBER OF NODES
9 - CLEAR TREE
0 - EXIT
ENTER: 5

Inorder traversal: 9 12 14 17 23 30 57 67 72 76
```

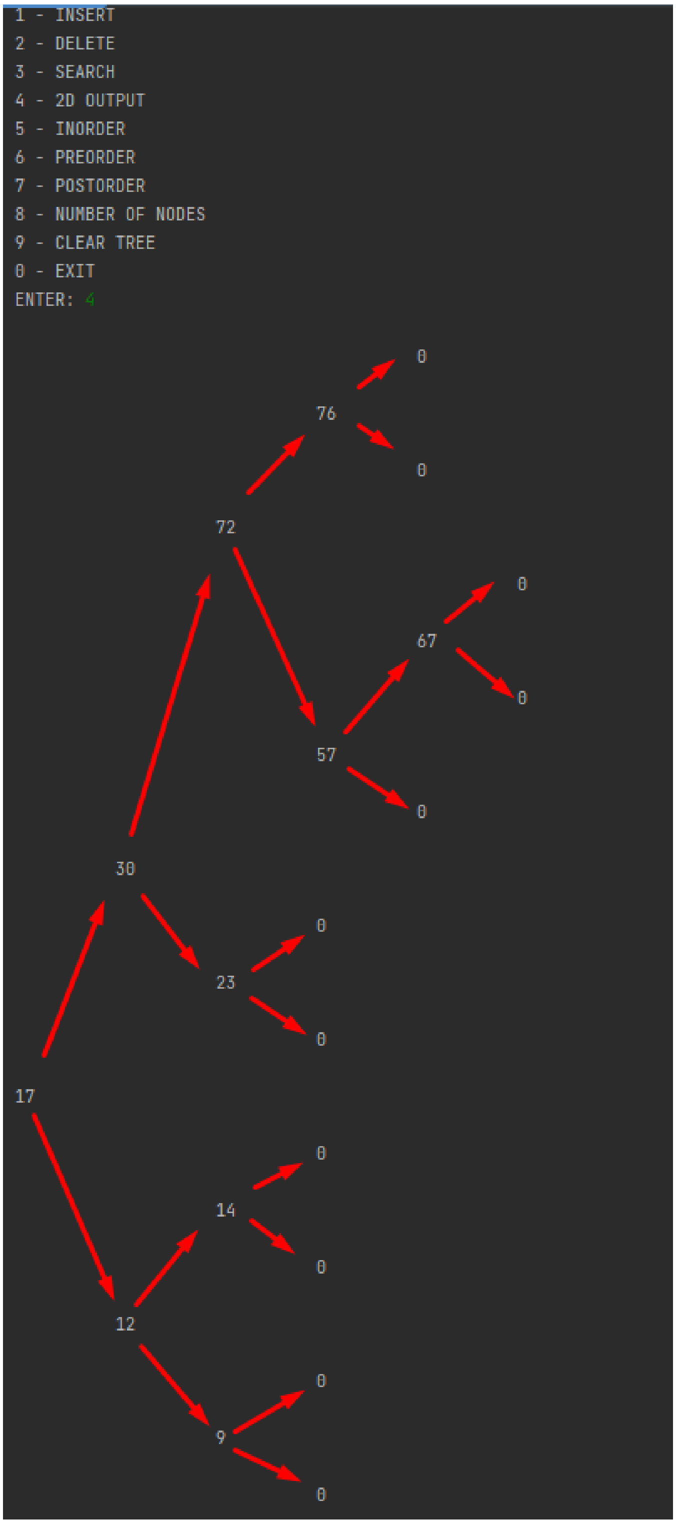
Скріншот 4.2 – Центрований обхід дерева

```
1 - INSERT
2 - DELETE
3 - SEARCH
4 - 2D OUTPUT
5 - INORDER
6 - PREORDER
7 - POSTORDER
8 - NUMBER OF NODES
9 - CLEAR TREE
0 - EXIT
ENTER: 6

Preorder traversal: 17 12 9 14 30 23 72 57 67 76
```

Скріншот 4.3 – Прямий обхід дерева

```
1 - INSERT
2 - DELETE
3 - SEARCH
4 - 2D OUTPUT
5 - INORDER
6 - PREORDER
7 - POSTORDER
8 - NUMBER OF NODES
9 - CLEAR TREE
0 - EXIT
ENTER: 7

Postorder traversal: 9 14 12 23 67 57 76 72 30 17
```

Скріншот 4.4 – Зворотний обхід дерева



Скріншот 4.5 – 2Д представлення дерева з коренем 17

Введемо нові значення: 5 4 7 6 8 11. І видалимо вузол 5, а потім 8:

```
                                    0

                           11

                                    0

                  8

                           0

         7

                           0

                  6

                           0

5

                  0

         4

                  0
```

Скріншот 4.6 – Поточне дерево

```
1 - INSERT
2 - DELETE
3 - SEARCH
4 - 2D OUTPUT
5 - INORDER
6 - PREORDER
7 - POSTORDER
8 - NUMBER OF NODES
9 - CLEAR TREE
0 - EXIT
ENTER: 8

Number of nodes: 6
```

Скріншот 4.7 – Поточна кількість вузлів

```
1 - INSERT
2 - DELETE
3 - SEARCH
4 - 2D OUTPUT
5 - INORDER
6 - PREORDER
7 - POSTORDER
8 - NUMBER OF NODES
9 - CLEAR TREE
0 - EXIT
ENTER: 2

Enter data for delete: 5

1 - INSERT
2 - DELETE
3 - SEARCH
4 - 2D OUTPUT
5 - INORDER
6 - PREORDER
7 - POSTORDER
8 - NUMBER OF NODES
9 - CLEAR TREE
0 - EXIT
ENTER: 2

Enter data for delete: 8
```

Скріншот 4.8 – Видалення вузлів

```
                                        0

                              11

                                        0

                    7

                              0

          6

                              0

                    4

                              0
```

Скріншот 4.9 – Результат

ВИСНОВОК

АА-Дерево – це одне з найшвидших бінарних дерев із простою реалізацією за рахунок своїх обмежень за структурою: у дереві не повинно бути лівих горизонтальних зв'язків та послідовних правих – ці дві умови забезпечують баланс усієї структури. Забезпечувати виконання обмежень беруть він дві функції – skew і split – видалення лівої горизонтального зв'язку і послідовних правих відповідно. Після вставки та видалення вузлів потрібно щоразу викликати skew та split. Інші функції не відрізняються від реалізації в звичайному Бінарному Дереві Пошуку.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Стаття на GeeksForGeeks, URL: https://www.geeksforgeeks.org/aa-trees-set-1-introduction/
2. Стаття та візуалізатор, URL: https://people.ksp.sk/~kuko/gnarley-trees/AAtree.html
3. Стаття на Habr, URL: https://habr.com/ru/post/110212/
4. Стаття про Червоно-Чорні Дерева на HappyCodders, URL: https://www.happycoders.eu/algorithms/red-black-tree-java/

ДОДАТОК А

Властивості Червоно-Чорного Дерева:

1. Кожен вузол забарвлений або червоний, або чорний колір (у структурі даних вузла з'являється додаткове поле – біт кольору).
2. Корінь забарвлений у чорний колір.
3. Листя (так звані NULL-вузли) пофарбовані в чорний колір.
4. Кожен червоний вузол повинен мати два чорні дочірні вузли.
   Слід зазначити, що з чорного вузла може бути чорні дочірні вузли.
   Червоні вузли як дочірні можуть мати лише чорні.
5. Шляхи від вузла до його листя повинні містити однакову кількість чорних вузлів (це чорна висота).

ДОДАТОК В

```
import BinaryTreePack.AATree;
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        AATree tree = new AATree();
        System.out.print("PROGRAM START");

        while (true) {
            System.out.print(
                    """
                    \n1 - INSERT
                    2 - DELETE
                    3 - SEARCH
                    4 - 2D OUTPUT
                    5 - INORDER
                    6 - PREORDER
                    7 - POSTORDER
                    8 - NUMBER OF NODES
                    9 - CLEAR TREE
                    0 - EXIT
                    ENTER:\040"""
            );
            int choice = scan.nextInt();
            switch (choice) {
                case 0 -> System.exit(1);
                case 1 -> {
                    System.out.print("\nEnter data for insert: ");
                    int data = scan.nextInt();
                    tree.insert(data);
                }
                case 2 -> {
                    System.out.print("\nEnter data for delete: ");
                    int data = scan.nextInt();
                    tree.delete(data);
                }
                case 3 -> {
                    System.out.print("\nEnter data for search: ");
                    int data = scan.nextInt();
                    if (tree.search(data)) {
                        System.out.printf("Node %d in the tree\n", data);
                    } else {
                        System.out.printf("Node %d not in the tree\n", data);
                    }
                }
                case 4 -> {
                    tree.print2D(tree.getRoot());
                }
                case 5 -> {
                    System.out.print("\nInorder traversal: ");
                    tree.inorderTraversal(tree.getRoot());
                }
                case 6 -> {
                    System.out.print("\nPreorder traversal: ");
                    tree.preorderTraversal(tree.getRoot());
                }
                case 7 -> {
                    System.out.print("\nPostorder traversal: ");
                    tree.postorderTraversal(tree.getRoot());
                }
```

```
                case 8 -> {
                    System.out.println("\nNumber of nodes: " +
tree.countNodes(tree.getRoot()));
                }
                case 9 -> {
                    tree.clear();
                    System.out.println("\nComplete!");
                }
            }
        }
    }
}
```

Main.java

```java
package BinaryTreePack;

public class AATree extends BaseBinaryTree {
    private static final Node nil = new Node();
    static final int COUNT = 10;

    public AATree() {
        root = nil;
    }

    public void clear() {
        root = nil;
    }

    public void insert(int data) {
        root = insertNode(data, root);
    }

    private Node insertNode(int data, Node node) {
        if (node == nil) {
            node = new Node(data, nil, nil);
        } else if (data < node.data) {
            node.left = insertNode(data, node.left);
        } else if (data > node.data) {
            node.right = insertNode(data, node.right);
        } else {
            return node;
        }

        node = skew(node);
        node = split(node);
        return node;
    }

    public void delete(int data) {
        root = deleteNode(data, root);
    }

    private Node deleteNode(int data, Node node) {
        if (node == nil) {
            return node;
        } else if (data > node.data) {
            node.right = deleteNode(data, node.right);
        } else if (data < node.data) {
            node.left = deleteNode(data, node.left);
        } else {
            if (node.left == nil && node.right == nil) {
                return nil;
            } else if (node.left == nil) {
                Node temp = node.right;
                while (temp.left.data != 0) {
                    temp = temp.left;
                }
                node.right = deleteNode(temp.data, node.right);
                node.data = temp.data;
            } else {
                Node temp = node.left;
                while (temp.right.data != 0) {
                    temp = temp.right;
                }
                node.left = deleteNode(temp.data, node.left);
                node.data = temp.data;
            }
```

```
        }

        node = decreaseLevel(node);
        node = skew(node);
        node.right = skew(node.right);

        if (node.right != nil) {
            node.right.right = skew(node.right.right);
        }
        node = split(node);
        node.right = split(node.right);
        return node;
    }

    private Node skew(Node node) {
        if (node == nil) {
            return nil;
        } else if (node.left == nil) {
            return node;
        } else if (node.left.level == node.level) {
            Node leftChild = node.left;
            node.left = leftChild.right;
            leftChild.right = node;
            return leftChild;
        } else {
            return node;
        }
    }

    private Node split(Node node) {
        if (node == nil) {
            return nil;
        } else if (node.right == nil || node.right.right == nil) {
            return node;
        } else if (node.level == node.right.right.level) {
            Node rightChild = node.right;
            node.right = rightChild.left;
            rightChild.left = node;

            rightChild.level = rightChild.level + 1;
            return rightChild;
        } else {
            return node;
        }
    }

    private Node decreaseLevel(Node node) {
        int shouldBe = Math.min(node.left.level, node.right.level) + 1;
        if (shouldBe < node.level) {
            node.level = shouldBe;
            if (shouldBe < node.right.level) {
                node.right.level = shouldBe;
            }
        }
        return node;
    }

    public int countNodes(Node rNode) {
        if (rNode == nil) {
            return 0;
        } else {
            int l = 1;
            l += countNodes(rNode.left);
```

```
            l += countNodes(rNode.right);
            return l;
        }
    }

    public boolean search(int data) {
        return searchNode(root, data);
    }

    private boolean searchNode(Node rNode, int data) {
        boolean found = false;
        while ((rNode != nil) && !found) {
            int rData = rNode.data;
            if (data < rData) {
                rNode = rNode.left;
            } else if (data > rData) {
                rNode = rNode.left;
            } else {
                found = true;
                break;
            }
            found = searchNode(rNode, data);
        }
        return found;
    }

    public void inorderTraversal(Node rNode) {
        if (rNode != nil) {
            inorderTraversal(rNode.left);
            System.out.printf("%d ", rNode.data);
            inorderTraversal(rNode.right);
        }
    }

    public void preorderTraversal(Node rNode) {
        if (rNode != nil) {
            System.out.printf("%d ", rNode.data);
            preorderTraversal(rNode.left);
            preorderTraversal(rNode.right);
        }
    }

    public void postorderTraversal(Node rNode) {
        if (rNode != nil) {
            postorderTraversal(rNode.left);
            postorderTraversal(rNode.right);
            System.out.printf("%d ", rNode.data);
        }
    }

    public void print2DUtil(Node root, int space) {
        if (root == null)
            return;

        space += COUNT;

        print2DUtil(root.right, space);

        System.out.print("\n");
        for (int i = COUNT; i < space; i++)
            System.out.print(" ");
        System.out.print(root.data + "\n");
```

```
        print2DUtil(root.left, space);
    }

    public void print2D(Node root) {
        print2DUtil(root, 0);
    }
}
```

AATree.java

```
package BinaryTreePack;

public class BaseBinaryTree implements BinaryTree {
    protected Node root;

    @Override
    public Node getRoot() {
        return root;
    };
}
```

BaseBinaryTree.java

```
package BinaryTreePack;

public interface BinaryTree {
    Node getRoot();
}
```

BinaryTree.java